

Modular Type-Safety Proofs using Dependant Types

Christopher Schwaab Jeremy G. Siek

University of Colorado at Boulder
{schwaab,jeremy.siek}@colorado.edu

Abstract

While methods of code abstraction and reuse are widespread and well researched, methods of proof abstraction and reuse are still emerging. We consider the use of dependent types for this purpose, introducing a completely mechanical approach to proof composition. We show that common techniques for abstracting algorithms over data structures naturally translate to abstractions over proofs. We first introduce a language composed of a series of smaller language components tied together by standard techniques from Malcom [2]. We proceed by giving proofs of type preservation for each language component and show that the basic ideas used in composing the syntactic data structures can be applied to their semantics as well.

1. Introduction

The POPLmark challenge is a set of common programming language problems meant to test the utility of modern proof assistants and techniques for mechanized metatheory. In response to this challenge, significant strides have been made in making it easier to mechanize the metatheory of programming languages, especially regarding variable binding [1]. However, little progress has been made in the direction of modularity: it is still difficult to separately develop the definitions and meta-theory of language fragments and then link the fragments together to obtain the definitions and meta-theory for a language composed of such fragments.

Dependent types have formed the foundation of a broad and rich range of type systems that allow values and types to be freely mixed. Programmers can express propositions as types viewed as sets, and proofs as objects viewed as inhabitants of those sets. This style of theorem proving suggests the use of familiar engineering abstractions as general solutions to questions about theorem proving. Rather than relying on semi-automated proof search such as Coq’s Ltac we propose a method of proof composition using simple abstractions whereby components are defined piecewise and “tied” together at the end using a wrapper datatype acting as a tagged union.

The method of language definition used is iterative. Components are defined separately from one another and are composable along with their proofs. Thus we would like for

separate language designers to be able to reuse one another’s work without the need for sophisticated proof search algorithms or with effort spent copying and pasting terms.

The language we present is one of simple expressions using Agda as the implementation language and proof assistant. We begin by defining a series of language syntaxes for sums, options, and arrays. We chose to include arrays because they not only can result in runtime errors requiring the inclusion of the *Option* type but like addition, they use the natural numbers, forcing consideration of how value types can be shared across otherwise isolated components. We continue by defining evaluation semantics and typing rules. The language is defined piecewise, each component is built in isolation alongside a proof of type preservation. We conclude with a presentation of how these components can be composed and a proof of type preservation for the combined language can be immediately derived from the component-wise proofs. The motivation for our technique is drawn from a solution to the expression problem where languages are defined as the disjoint sum of smaller languages by removing explicit recursion. We show that this idea can be recast from types and terms, to proofs.

2. A Review of the Expression Problem

When modeling a problem with a functional flavor often the natural solution emerges as several recursive cases handled by some helper functions. The expression problem states that this type of solution presents us with a choice: we may ordain our data structure forever unchanging, making it easy to add new functions without changing the program; or we may leave our data structure open, making it difficult to extend the original program with new functions.

While many solutions to the expression problem have been proposed over the years, here we make use of the method described by Malcom [2] which generalizes recursion operators such as fold, from lists to polynomial types. The problem we encounter arises as a result of algebraic data types being *closed*: once the type has been declared, no new constructors for the type may be added without amending the original declaration and the solution presented lies at the heart of our work. The idea is simply to remove immediate recursion and split a monolithic datatype into components to be later collected under the umbrella of a tagged union.

Throughout this paper we will work with a simple evaluator over natural numbers and basic arithmetic operators; in Agda we might first consider

```
data Expr+ : Set where
  atom  : ℕ → Expr+
  _+_   : Expr+ → Expr+ → Expr+
```

[Copyright notice will appear here once ‘preprint’ option is removed.]

This definition has the advantage of being direct and simple, however a problem lies within the explicit recursion; notice that when later extending expressions with arrays and option types we can make no reuse of `Expr+` due to the closed nature of algebraic data types. To extend `Expr+` we must define a whole new data type, as in the following definition of `MonolithicExpr`.

```

data MonolithicExpr : Set where
  atom : ℕ → MonolithicExpr
  esome : MonolithicExpr → MonolithicExpr
  enone : MonolithicExpr
  nil [] : MonolithicExpr
  _!!_ : MonolithicExpr → MonolithicExpr
  → MonolithicExpr
  _[_]_ := _ : MonolithicExpr → MonolithicExpr
  → MonolithicExpr
  _+_ : MonolithicExpr → MonolithicExpr
  → MonolithicExpr
  fromExpr+ : Expr+ → MonolithicExpr
  fromExpr+ (atom n) = atom n
  fromExpr+ (n + m) = fromExpr+ n + fromExpr+ m

```

Suppose instead we begin with polymorphic definitions such as the following.

```

data Expr+2 (A : Set) : Set where
  _+_ : A → A → Expr+2 A
data Expr []2 (A : Set) : Set where
  nil [] : Expr []2 A
  _!!_ : Expr []2 A
  _[_]_ := _ : A → A → A → Expr []2 A
data ExprOption (A : Set) : Set where
  esome : ℕ → ExprOption A
  enone : ℕ → ExprOption A
data Lit (A : Set) : Set where
  atom : ℕ → Lit A

```

We then introduce recursion as follows, combining components as a disjoint sum, written $- \uplus -$ in Agda.

```

data RecExpr : Set where
  expr : Lit RecExpr
  ⊔ Expr+2 RecExpr
  ⊔ Expr []2 RecExpr
  ⊔ ExprOption RecExpr
  → RecExpr

```

More generally, this type of data can be captured using a “categorical approach” where recursion is introduced as the fixed point of a functor:

```

data μ_ (F : Set → Set) : Set where
  inn : F (μ F) → μ F
  Expr' = λ (A : Set) → Lit A ⊔ Expr+2 A
  ⊔ Expr []2 A
  ⊔ ExprOption A
  Expr = μ Expr'

```

It is easy to see that this new type is equivalent to `MonolithicExpr` up to isomorphism

$$\begin{aligned}
 Expr &= \mu Expr' \\
 &= Expr' (\mu Expr') \\
 &= Expr' Expr \\
 &\cong atom \\
 &\quad | esome | enone \\
 &\quad | Expr + Expr \\
 &\quad | nil[] | -!!- | -[_]- := -
 \end{aligned}$$

2.1 Functors and Agda

The functor F , passed into $\mu-$ above, serves as the key abstraction allowing us to represent expressions as least fixed points. Functors are a special mapping defined over both types and functions satisfying the so called *functor laws*; a functor F

1. assigns to each type A , a type $F A$
2. assigns to each function $f : A \rightarrow B$, a function $\text{map } f : F A \rightarrow F B$

such that

1. identity is preserved: $\text{map id} = \text{id}$, and
2. when $f \circ g$ is defined: $\text{map } (f \circ g) = \text{map } f \circ \text{map } g$.

One familiar example is the *List* functor mapping each type A to $\text{List } A$ and each function $f : A \rightarrow B$ to $\text{map } f : \text{List } A \rightarrow \text{List } B$ which applies f to each element of a list. Here we define the least fixed point over a restricted class of functors called the *polynomial functors*. Polynomial functors are a subset roughly equivalent to the more familiar algebraic polynomials,

$$\sum_{n \in \mathbb{N}} A_n X^n$$

where addition is disjoint sum and multiplication is cartesian product. In Agda, Ulf Norell[4] expresses this class as a datatype *Functor* along with an interpretation as a set $[-]$

```

infixl 6 _⊕_
infixr 7 _⊗_
data Functor : Set1 where
  X : Functor
  A : Set → Functor
  _⊕_ : Functor → Functor → Functor
  _⊗_ : Functor → Functor → Functor
  [-] : Functor → Set → Set
  [X] B = B
  [A C] B = C
  [F ⊕ G] B = [F] B ⊔ [G] B
  [F ⊗ G] B = [F] B × [G] B

```

with least fixed point

```

data μ_ (F : Functor) : Set where
  inn : [F] (μ F) → μ F

```

Then to reexpress *Expr* as a polynomial functor we use sum $- \oplus -$ to define cases within a type, and product $- \otimes -$ to represent arguments of a particular case

```

Option1 : Functor
Option1 = X ⊕ A ⊤

```

```

Array1 : Functor
Array1 = X ⊕ X ⊕ X ⊕ A T ⊕ X ⊕ X
Sum1 : Functor
Sum1 = X ⊗ X
F1 : Functor
F1 = A N ⊕ Option1 ⊕ Sum1 ⊕ Array1
E1 : Set
E1 = μ F1

```

Unfolding E_1 yields the same value calculated above—as we should hope!

```

E1 = μ F1
    = [A N ⊕ Option1 ⊕ Sum1 ⊕ Array1]
    = [A N] (μ F1)
    ⊔ [Option1] (μ F1)
    ⊔ [Sum1] (μ F1)
    ⊔ [Array1] (μ F1)
    = N
    ⊔ (μ F1) × T
    ⊔ (μ F1) × (μ F1)
    ⊔ (μ F1) × (μ F1) ⊔ (μ F1) ⊔ T ⊔ (μ F1) ⊔ (μ F1)

```

What do values in E_1 look like? Written directly they appear nonsensical, consider $6 + 7$

```

the-sum : E1
the-sum = inn (inj1 (inj2 (
  (inn (inj1 (inj1 (inj1 6))))
  , (inn (inj1 (inj1 (inj1 7)))))))

```

Notice here the role that the injections and *inn* functions play. Traditionally we would provide a unique name for each branch in an algebraic datatype, however here we only have two names *inj₁* and *inj₂* so we instead rely on nesting to create unique prefixes. Once we have tagged a value we must give it a well known type so that parent expressions can expect a common child type, this is the role of *inn*. Although cumbersome we can hide much of this complexity provided the right abstractions

```

the-sum' : E1
the-sum' = nat1 6 +1 nat1 7
  where nat1 : N → E1
        nat1 = inn ∘ inj1 ∘ inj1 ∘ inj1
        _+1_ : E1 → E1 → E1
        e1 +1 e2 = inn (inj1 (inj2 (e1, e2)))

```

3. Syntax and Evaluation Semantics

We are now ready to define a simple language and its operational semantics. The language is small including just sums, an option type, and an array with assignment and lookup. In Agda, the unit type is written \top and has only one member: *tt*. \top is used to represent constructors that take no arguments such as *nil*, the empty list.

```

Option : Functor
Option = X ⊕ A T
Array : Functor
Array = X ⊗ X ⊗ X ⊕ A T ⊕ X ⊗ X
Sum : Functor
Sum = X ⊗ X
FExpr : Functor
FExpr = A N ⊕ Option ⊕ Sum ⊕ Array

```

```

Expr : Set
Expr = μ FExpr

```

What do each of these definitions mean? The maybe type has two constructors: *some*, which wraps a single expression; and *none* taking no arguments. We define more descriptive constructors for tagging these two types of values

```

none1 : Expr
none1 = inn (inj1 (inj1 (inj2 (tt))))
some1 : Expr → Expr
some1 = inn ∘ inj1 ∘ inj1 ∘ inj2 ∘ inj1

```

Giving a convenient constructor for $- + -$ is similarly straightforward

```

enat : N → Expr
enat = inn ∘ inj1 ∘ inj1 ∘ inj1
_+_ : Expr → Expr → Expr
e1 + e2 = inn (inj1 (inj2 (e1, e2)))

```

and to define arrays we have assignment taking an array, an index, and a value to assign at that index; *nil*, the empty array; and *lookup* which accepts an array and an index

```

_[-]_ :=1 _ : Expr → Expr → Expr → Expr
a [i] :=1 e = inn (inj2 (inj1 (inj1 (a, i, e))))
nil1 : Expr
nil1 = inn (inj2 (inj1 (inj2 (tt))))
_!_1 : Expr → Expr → Expr
a !1 i = inn (inj2 (inj2 (a, i)))

```

So far the definition of our syntax has used fairly standard techniques but we have failed to give any sort of meaning to these expressions. We first define a monolithic static and dynamic semantics for this language, then show how to modularize their definition later in this section. Figure 1c defines a simple set of typing rules using metavariables *e* to range over expressions and *n* to range over values; Figure 1b gives a small step operational semantics.

While Agda is expressive enough to implement these rules, directly and indeed they are nearly a direct reflection of that implementation, recall that our goal is to create several independent languages each carrying their own semantics. We begin by defining monolithic semantics for *Expr* and proceed to determine points of failure and to dissect the definition into independent constituents. To simplify things we define our notion of *Type* as a closed ADT

```

data Type : Set where
  TArray : Type
  TOption : Type
  TNat : Type

```

and here is the definition of the monolithic type system and evaluation relation in Agda.

```

data Welltyped : Expr → Type → Set1 where
  ok-value : {n : N} → Welltyped (enat n) TNat
  ok-sum : {e1 e2 : Expr}
    → Welltyped e1 TNat → Welltyped e2 TNat
    → Welltyped (e1 + e2) TNat
  ok-nil : Welltyped nil1 TArray
  ok-lookup : {a e : Expr}
    → Welltyped a TArray
    → Welltyped e TNat
    → Welltyped (a !1 e) TOption
  ok-ins : {a e n : Expr}
    → Welltyped a TArray

```

$$\begin{array}{l}
- \dot{+} - \in Expr \rightarrow Expr \rightarrow Expr \\
n \in \mathbb{N} \in Expr \\
-[-] := - \in Expr \rightarrow Expr \rightarrow Expr \rightarrow Expr \\
-!- \in Expr \rightarrow Expr \rightarrow Expr \\
nil \in Expr \\
\text{(a) Syntax}
\end{array}
\qquad
\begin{array}{l}
(\text{stepl}+) \frac{e_1 \longrightarrow e'_1}{e_1 \dot{+} e_2 \longrightarrow e'_1 \dot{+} e_2} \\
(\text{stepr}+) \frac{e_2 \longrightarrow e'_2}{n_1 \dot{+} e_2 \longrightarrow n_1 \dot{+} e'_2} \\
(\text{sum}) \frac{}{n_1 \dot{+} n_2 \longrightarrow n_1 + n_2} \\
(\text{stepi}) \frac{e \longrightarrow e'}{a!e \longrightarrow a!e'} \\
(\text{lookup}) \frac{}{a!n \longrightarrow L[a, n]} \\
\text{(b) Evaluation Semantics}
\end{array}$$

$$\begin{array}{l}
(\text{ok-value}) \frac{}{n : Nat} \\
\text{(c) Value Typing}
\end{array}
\qquad
\begin{array}{l}
(\text{ok-sum}) \frac{e_1 : Nat \quad e_2 : Nat}{e_1 \dot{+} e_2 : Nat} \\
\text{(d) Sum Typing}
\end{array}$$

$$\begin{array}{l}
(\text{ok-nil}) \frac{}{nil : Array} \\
(\text{ok-lookup}) \frac{a : Array \quad e : Nat}{a!e : Option} \\
(\text{ok-ins}) \frac{a : Array \quad n : Nat \quad e : Nat}{a[n] := e : Array} \\
\text{(e) Array Typing}
\end{array}$$

```

→ Welltyped e TNat
→ Welltyped n TNat
→ Welltyped (a [n] := e) TArray
infix 2 _→E_
data _→E_ : Expr → Expr → Set where
  stepl : {e1 e1' e2 : Expr}
    → e1 →E e1'
    → e1 \dot{+} e2 →E e1' \dot{+} e2
  stepr : {n1 : \mathbb{N}} {e2 e2' : Expr}
    → e2 →E e2'
    → enat n1 \dot{+} e2 →E enat n1 \dot{+} e2'
  sum : {n1 n2 : \mathbb{N}}
    → enat n1 \dot{+} enat n2 →E enat (n1 + \mathbb{N} n2)
  stepi : {e e' a : Expr}
    → e →E e'
    → a !_1 e →E a !_1 e'
  lookup : {a n : Expr}
    → a !_1 n →E L[a, n]_1

```

The function $L[-, -]_1$ is the lookup function that evaluates to some a_n when a_n has been defined and *none* otherwise.

Notice that we currently do not restrict the values of n enough in the *ok-ins* rule; our typing rules require that n be a value while in Agda we have only required it be an expression. Some notion of value is needed and a common solution is to add a tag *Value* to the *Expr* type and pattern match; here *Value* is called $[AN]$ and in a dependently typed context we might then define a predicate over *Value*. However because the sum type has only one type of value, a number, it is simpler to use *enat* directly.

This method for defining semantics is common with the advantage of being direct and concise, but similar to our first implementation of *Expr+* and *MonolithicExpr* above: there is no simple mechanism for code reuse. The answer is again to delay recursion.

3.1 Dissecting the Step Relation

In order to modularize the evaluation rules we define a separate step relation for each functor making up our *Expr* type. First note that $-\dot{+}-$ doesn't make use of *how* the step from e_1 to e_2 occurs so we can factor this top-level relation

```

data  $\_ \longrightarrow^+ \_$  {  $\_ \longrightarrow \_ : \text{Expr} \rightarrow \text{Expr} \rightarrow \text{Set}$  }
  :  $\text{Expr} \rightarrow \text{Expr} \rightarrow \text{Set}$  where
stepl : {  $e_1 e_1' e_2 : \text{Expr}$  }
   $\rightarrow e_1 \longrightarrow e_1' \rightarrow e_1 \dot{+} e_2 \longrightarrow^+ e_1' \dot{+} e_2$ 
stepr : {  $n_1 : \mathbb{N}$  } {  $e_2 e_2' : \text{Expr}$  }
   $\rightarrow e_2 \longrightarrow e_2'$ 
   $\rightarrow \text{enat } n_1 \dot{+} e_2 \longrightarrow^+ \text{enat } n_1 \dot{+} e_2'$ 
sum : {  $n_1 n_2 : \mathbb{N}$  }
   $\rightarrow \text{enat } n_1 \dot{+} \text{enat } n_2 \longrightarrow^+ \text{enat } (n_1 + \mathbb{N} n_2)$ 

```

While this is better there is still an undesirable reference to the datatype *Expr*. Applying the same factorization here to the underlying functor requires parametrization by two extra coercion functions, these are the $\dot{+}$ and *enat* functions defined previously. The new names *lift*⁺ and *liftN* used here are meant to imply that a subtype is being “lifted” into its supertype

```

data  $\_ \longrightarrow^+ \_$  {  $E : \text{Functor}$  } {  $\_ \longrightarrow \_ : \mu E \rightarrow \mu E \rightarrow \text{Set}$  }
  { lift+ : [Sum] ( $\mu E$ )  $\rightarrow \mu E$  } { liftN :  $\mathbb{N} \rightarrow \mu E$  }
  :  $\mu E \rightarrow \mu E \rightarrow \text{Set}$  where
stepl : {  $e_1 e_1' e_2 : \mu E$  }
   $\rightarrow e_1 \longrightarrow e_1' \rightarrow \text{lift}^+ (e_1, e_2) \longrightarrow^+ \text{lift}^+ (e_1', e_2)$ 
stepr : {  $n_1 : \mathbb{N}$  } {  $e_2 e_2' : \mu E$  }
   $\rightarrow e_2 \longrightarrow e_2'$ 
   $\rightarrow \text{lift}^+ (\text{liftN } n_1, e_2) \longrightarrow^+ \text{lift}^+ (\text{liftN } n_1, e_2')$ 
sum : {  $n_1 n_2 : \mathbb{N}$  }
   $\rightarrow \text{lift}^+ (\text{liftN } n_1, \text{liftN } n_2) \longrightarrow^+ \text{liftN } (n_1 + \mathbb{N} n_2)$ 

```

Unfortunately this definition falls short too. When we lift terms into the expression type μE , Agda “forgets” the constituents e_1 and e_2 —in turn we lose the ability to reason about these distinct components of the sums e_n and e'_n . This later becomes a problem when, for example, attempting to abstract the welltyping relation.

An intelligent human can peel away *lift*⁺ and see that the terms e_1 and e_2 in $_ \longrightarrow _$ and $_ \longrightarrow^+ _$ are the same because *lift*⁺ is *injective*. However Agda is unconvinced, and rightfully so, for it does not require a particularly great deal of ingenuity to find a counterexample, consider taking $E = F\text{Expr}$ so that $\mu E = \text{Expr}$

```

forgetful-lift+ : [Sum] Expr  $\rightarrow$  Expr
forgetful-lift+ ( $e_1, e_2$ ) = enat 0

```

The problem is that our abstraction is too general. What we require is a proof that [Sum](μE) and \mathbb{N} are subtypes of the top-level expression datatype μE . The solution to the problem is drawn from the notion of a categorical subobject.

We proceed by delaying application of injections and view the objects as injectable, existential terms. The importance of this approach is two-fold: firstly this allows us to take inverses of lift functions while we are secondly able to retain the perspective of operating on a single type μE .

4. Lazy Coercions

A subobject of a type T is a left invertible function with codomain T , *lift* : $S \hookrightarrow T$. Being restricted to polynomial functors, we know that all our subobjects *lift* : $S \rightarrow \mu E$ will be some composition of *inn*, *inj*₁ and *inj*₂ so a proof that S is a subtype of μE is merely a description of which direction to move at each point in a disjoint sum

```

infix 3  $\_ \text{Contains}$   $\_$ 
data  $\_ \text{Contains}$   $\_$  : Functor  $\rightarrow$  Functor  $\rightarrow \text{Set}_1$  where
  refl : {  $F : \text{Functor}$  }

```

```

   $\rightarrow F \text{Contains } F$ 
left : {  $A B F : \text{Functor}$  }
   $\rightarrow F \text{Contains } A \oplus B \rightarrow F \text{Contains } A$ 
right : {  $A B F : \text{Functor}$  }
   $\rightarrow F \text{Contains } A \oplus B \rightarrow F \text{Contains } B$ 

```

Now we can define containment on a functor’s interpretation as a set

```

infix 3  $\_ \rightsquigarrow \_$ 
data  $\_ \rightsquigarrow \_$  : Set  $\rightarrow$  Set  $\rightarrow \text{Set}_1$  where
  inj : {  $\bar{F} A : \text{Functor}$  }
   $\rightarrow F \text{Contains } A \rightarrow [A] (\mu F) \rightsquigarrow (\mu F)$ 

```

with conversion functions defined as

```

upcast :  $\forall \{ F A \} \rightarrow F \text{Contains } A \rightarrow [A] (\mu F) \rightarrow \mu F$ 
upcast refl = inn
upcast (left t) = upcast t  $\circ$  inj1
upcast (right t) = upcast t  $\circ$  inj2
apply : {  $A B : \text{Set}$  }  $\rightarrow (A \rightsquigarrow B) \rightarrow A \rightarrow B$ 
apply (inj t) = upcast t

```

Recall the two goals we had in mind. We first wished to take the inverse of a lift function to gain access to its arguments, in the case of $\dot{+}$ —these were e_1 and e_2 . By representing an injection as a delayed application of a subobject—because the constructor’s arguments are stored as a part of the coercion—finding left inverses will become a trivial case of pattern matching. To delay function application allowing Agda to effectively peel away the *lift* functions we define a *LazyCoercion* datatype from type A to B representing the *intention* of coercing an object $a \in A$ while treating it at the type-level as B . A lazy coercion is then an injection $A \rightsquigarrow B$ along with an object in A

```

data LazyCoercion : Set  $\rightarrow$  Set1 where
  inj : {  $A B : \text{Set}$  }  $\rightarrow (A \rightsquigarrow B) \rightarrow A \rightarrow \text{LazyCoercion } B$ 
  coerce : {  $B : \text{Set}$  }  $\rightarrow \text{LazyCoercion } B \rightarrow B$ 
  coerce (inj f e) = apply f e

```

Our second goal was to operate on objects of a single type. Why is this the case? Recall that the type of our step relation is indexed by two expressions: ($e_1 : \text{Expr}$) \longrightarrow_E ($e_2 : \text{Expr}$). We should expect the same of the final abstraction over step relations because it cannot easily name the underlying type of its indexing expressions. Instead we have packaged the indices as *existentials* which are viewed as the type B .

We seem to be close to a modular step relation $_ \longrightarrow^+ _$, defining at each point another level of abstraction to delay immediate application. To modularize datatypes, recursion is delayed and types are viewed as polynomial functors, then to modularize step relations, evaluation is parametrized and expression upcasts are delayed by viewing them as an intention.

5. Defining a Modular Step Relation

Attempting again to define a step relation for addition we find very little has changed

```

data  $\_ \longrightarrow^+ \_$  {  $E : \text{Functor}$  }
  {  $\_ \longrightarrow \_ : \mu E \rightarrow \mu E \rightarrow \text{Set}_1$  }
  { lift+ : [Sum] ( $\mu E$ )  $\rightarrow \mu E$  }
  { liftN :  $\mathbb{N} \rightarrow \mu E$  }
  : LazyCoercion ( $\mu E$ )  $\rightarrow$  LazyCoercion ( $\mu E$ )  $\rightarrow \text{Set}_1$ 
  where
stepl : {  $e_1 e_1' e_2 : \mu E$  }

```

```

→ e1 → e'1'
→ inj lift+ (e1, e2) →+ inj lift+ (e'1, e2)
stepr : {e1 e2 e'2 : μ E}
→ e2 → e'2'
→ inj lift+ (e1, e2) →+ inj lift+ (e1, e'2')
stepv : {n m : ℕ}
→ inj lift+ (apply liftℕ n, apply liftℕ m)
→+ inj liftℕ (n + ℕ m)

```

It appears we've littered an otherwise simple definition with *inj* but we've replaced our arbitrary arrows with objects having constructors we can match on. Using the above techniques we can modularize the welltyping relation over sums for free

```

data WtSum {E : Functor}
{Wt : μ E → Type → Set1}
{lift+ : [Sum] (μ E) → μ E}
: LazyCoercion (μ E) → Type → Set1 where
ok-sum : {e1 e2 : μ E}
→ Wt e1 Tℕat → Wt e2 Tℕat
→ WtSum (inj lift+ (e1, e2)) Tℕat

```

The above definitions nearly wrote themselves. The simplicity comes from the fact we are just abstracting as many terms as possible, keeping in mind we can fill them in naturally later because the abstraction is so general there are few options available.

5.1 Arrays

We proceed by defining the step and welltypedness relations on arrays that can be combined with the relations on sums. The definitions for evaluation and welltypedness should look similar to those for sums — \rightarrow^+ —.

```

data _→_ [] {E : Functor}
{ _→_ : μ E → μ E → Set1 }
{liftA : [Array] (μ E) → μ E}
{liftℕ : ℕ → μ E}
{liftO : [Option] (μ E) → μ E}
: LazyCoercion (μ E) → LazyCoercion (μ E) → Set1
where
stepr : {e e' a : μ E} → e → e'
→ inj liftA (a ! e) → [] inj liftA (a ! e')
lookup : {a : [Array] (μ E)} {n : ℕ}
→ inj liftA (apply liftA a ! apply liftℕ n)
→ [] inj liftO L[a, n]

```

To define the typing relation we again follow the format of *WtSum* above and we are done.

```

data WtArray {E : Functor}
{Wt : μ E → Type → Set1}
{liftA : [Array] (μ E) → (μ E)}
{liftℕ : ℕ → μ E}
: LazyCoercion (μ E) → Type → Set1 where
ok-nil : WtArray (inj liftA nil) TArray
ok-ins : {a e n : μ E}
→ Wt a TArray → Wt e Tℕat → Wt n Tℕat
→ WtArray (inj liftA (a [n] := e)) TArray
ok-lookup : {e a : μ E}
→ Wt a TArray → Wt e Tℕat
→ WtArray (inj liftA (a ! e)) TOption

```

6. Proving Type Preservation

The type preservation lemma states that if a term is well-typed and can step, then the type of the term is preserved after evaluation

$$e \rightarrow e' \wedge e : T \Rightarrow e' : T \quad (\text{type-preservation})$$

Prior to considering how type preservation might look for each of the previously defined components we should review what type preservation looks like for the *MonolithicExpr* language. The proof is standard, proceeding by structural induction on the shape of the welltyping tree.

```

preservation-MonolithicExpr : ∀ {e e'} {τ}
→ e → C e'
→ WtMonolithicExpr e τ
→ WtMonolithicExpr e' τ
preservation-MonolithicExpr (stepl ste1) (ok-sum wte1 wte2)
= ok-sum (preservation-MonolithicExpr ste1 wte1) wte2
preservation-MonolithicExpr (stepr ste2) (ok-sum wte1 wte2)
= ok-sum wte1 (preservation-MonolithicExpr ste2 wte2)
preservation-MonolithicExpr
(stepv {n} {m}) (ok-sum wtn wtm)
= ok-nat (n + ℕ m)
preservation-MonolithicExpr (stepl ste) (ok-lookup wta wte)
= ok-lookup wta (preservation-MonolithicExpr ste wte)
preservation-MonolithicExpr
(lookup {a} {n}) (ok-lookup wta wtn)
= proj2 LC[a, n]

```

There are three items worth noting here: the first is the use of the function $LC[-, -] : \text{MonolithicExpr} \rightarrow \mathbb{N} \rightarrow \exists e. \text{WtMonolithicExpr } e \text{ TOption}$ which we have assumed produces a pair with first component an expression and second component a proof that the expression is a welltyped option; the second is that recursion acts as our induction hypothesis; and finally that Agda is smart enough to notice there is only a single possible welltyping constructor for each step constructor—in Agda all functions are total.

We should expect the modular type preservation lemmas to look similar because there is little global knowledge involved. The induction hypothesis and values aside, each case is “contained within its own world” in the sense that each evaluation rule relies only on the fact that subterms are well-typed but ignoring the *reason* they are welltyped. To show type preservation for sums we might start with

```

preservation-Sum1 : {τ : Type} {E : Functor}
{e e' : LazyCoercion (μ E)}
→ e →+ e'
→ WtSum e τ
→ WtSum e' τ
preservation-Sum1
(stepl {e1} {e'1'} {e2} ste1) (ok-sum wte1 wte2) = *

```

however recall that \rightarrow^+ requires the top-level step relation and proof that *E* contains both sums and naturals. There is a second mistake in writing preservation this way—we would like to show that *e'* is welltyped in the expression language, not just necessarily in the modular sum language, this reflects our desire to expose as little about each component as possible. A second formulation might then begin as follows but we again fail.

```

preservation-Sum2 : {τ : Type}
{E : Functor}
{ _→_ : μ E → μ E → Set1 }

```

```

{lift+ : [Sum] (μ E) → μ E}
{liftN : N → μ E}
{Wt : μ E → Type → Set1}
{e e' : LazyCoercion (μ E)}
→ _ →+ _ {E} { _ → _ } {lift+} {liftN} e e'
→ WtSum {E} {Wt} {lift+} e τ
→ Wt (coerce e') τ
preservation-Sum2 (stepl ste1) (ok-sum wte1 wte2)
= * (ok-sum * wte2)
preservation-Sum2 (stepr ste1) (ok-sum wte1 wte2)
= * (ok-sum wte1 *)
preservation-Sum2 stepv (ok-sum wte1 wte2)
= * (n + N m)

```

It seems we're only missing two pieces: we need to be able to lift welltyped sums and naturals into *Wt*; and we need some way of expressing the induction hypothesis which states that because e_1 is welltyped and stepped, e'_1 is welltyped too. The induction hypothesis is slightly stranger than was the case in our *MonolithicExpr*'s because we know e_1 and e'_1 are welltyped despite the fact that they are any expressions, not necessarily just sums. This motivates our solution which takes the induction hypothesis as an explicit assumption.

```

preservation-Sum : {τ : Type}
{E : Functor}
{ _ → _ : μ E → μ E → Set1 }
{lift+ : [Sum] (μ E) → μ E}
{liftN : N → μ E}
{Wt : μ E → Type → Set1}
{a b : LazyCoercion (μ E)}
→ ((n : N) → Wt (apply liftN n) TNat)
→ (∀ {δ} {e}
→ WtSum {E} {Wt} {lift+} (inj lift+ e) δ
→ Wt (apply lift+ e) δ)
→ (∀ {δ} {e e'} → e → e' → Wt e δ → Wt e' δ)
→ _ →+ _ {E} { _ → _ } {lift+} {liftN} a b
→ WtSum {E} {Wt} {lift+} a τ
→ Wt (coerce b) τ
preservation-Sum wtnat wt IH
(stepl ste1) (ok-sum wte1 wte2)
= wt (ok-sum (IH ste1 wte1) wte2)
preservation-Sum wtnat wt IH
(stepr ste2) (ok-sum wte1 wte2)
= wt (ok-sum wte1 (IH ste2 wte2))
preservation-Sum wtnat wt IH
(stepv {n} {m}) (ok-sum wte1 wte2)
= wtnat (n + N m)

```

We are pleased with how similar this is to the original, monolithic formulation. Notice again that the solution was to factor out assumptions about the outside world similar to the previous abstractions. Proving type preservation for arrays is similarly natural:

```

preservation-Array : {τ : Type}
{E : Functor}
{ _ → _ : μ E → μ E → Set1 }
{liftA : [Array] (μ E) → (μ E)}
{liftN : N → μ E}
{liftO : [Option] (μ E) → (μ E)}
{Wt : μ E → Type → Set1}
{a b : LazyCoercion (μ E)}
→ ((m : [Option] (μ E)) → Wt (apply liftO m) TOption)
→ (∀ {δ} {e}

```

```

→ WtArray {E} {Wt} {liftA} {liftN} (inj liftA e) δ
→ Wt (apply liftA e) δ)
→ (∀ {δ} {e e'} → e → e' → Wt e δ → Wt e' δ)
→ _ → [ ] _ {E} { _ → _ } {liftA} {liftN} {liftO} a b
→ WtArray {E} {Wt} {liftA} {liftN} a τ
→ Wt (coerce b) τ
preservation-Array wtopt wt IH
(stepi ste) (ok-lookup wta wte)
= wt (ok-lookup wta (IH ste wte))
preservation-Array wtopt wt IH
(lookup {a} {n}) (ok-lookup wta wte)
= wtopt L [ a, n ]

```

It would seem we're nearly done and the final pieces should be entirely guided by the selected abstractions. The *lift* functions each have a unique solution:

```

lift+ : [Sum] Expr → Expr
lift+ = inj (right (left (refl)))
liftN : N → Expr
liftN = inj (left (left (left refl)))
liftO : [Option] Expr → Expr
liftO = inj (right (left (left refl)))
liftA : [Array] Expr → Expr
liftA = inj (right refl)

```

But how should we define welltypedness for *Expr*? Again the notion of what it means to be welltyped has already been defined and we simply need to “tie the knot” as *RecExpr* did above

```

data WtExpr : Expr → Type → Set1 where
lift-wt-nat : (n : N) → WtExpr (apply liftN n) TNat
lift-wt-option : (m : [Option] Expr)
→ WtExpr (apply liftO m) TOption
lift-wt-sum : {τ : Type} {e : [Sum] Expr}
→ WtSum {FExpr} {WtExpr} {lift+} (inj lift+ e) τ
→ WtExpr (apply lift+ e) τ
lift-wt-array : {τ : Type} {e : [Array] Expr}
→ WtArray {FExpr} {WtExpr} {liftA} {liftN}
(inj liftA e) τ
→ WtExpr (apply liftA e) τ

```

To define a step relation on *Expr*, $- \rightarrow -$ we provide a similar wrapping for each language component

```

data _ → _ : Expr → Expr → Set1 where
step+ : {e : [Sum] Expr} {e' : LazyCoercion Expr}
→ _ →+ _ {FExpr} { _ → _ } {lift+} {liftN}
(inj lift+ e) e'
→ apply lift+ e → coerce e'
step [ ] : {e : [Array] Expr} {e' : LazyCoercion Expr}
→ _ → [ ] _
{FExpr} { _ → _ } {liftA} {liftN} {liftO}
(inj liftA e) e'
→ apply liftA e → coerce e'

```

The only piece remaining is to prove type preservation. We begin in the same way we have for each of the previous proofs using the step relation's constructors as a guide. The type signature should not have changed

```

preservation : {e e' : Expr} {τ : Type}
→ e → e' → WtExpr e τ → WtExpr e' τ

```

and there are two cases *step⁺* and *step []*; moreover we should expect to merely apply *preservation-** to each case,

supplying the necessary lift functions and the induction hypothesis. This is indeed the case:

```

preservation (step+ ste) (lift-wt-sum wts)
  = preservation-Sum lift-wt-nat lift-wt-sum
    preservation ste wts
preservation (step [] ste) (lift-wt-array wta)
  = preservation-Array lift-wt-option lift-wt-array
    preservation ste wta

```

Having shown type preservation it is interesting to see the similarity between how terms are shown to be welltyped and to evaluate and how the terms are expressed in $\mu FExpr$. Recall that each term in $Expr$ is wrapped by a tag—given by inj_1 and inj_2 —and the constructor inn plays the role of recursion. To reiterate consider the convenience functions,

```

nilE : Expr
nilE = inn (inj2 (inj1 (inj2 tt)))
nat : ℕ → Expr
nat n = inn (inj1 (inj1 (inj1 n)))
_ [ _ ] = _ : Expr → Expr → Expr → Expr
a [ n ] = e = apply liftA (a [ n ] := e)
_ ! E _ : Expr → Expr → Expr
a ! E n = apply liftA (a ! n)
_ + E _ : Expr → Expr → Expr
e1 + E e2 = apply lift+ (e1, e2)

```

we may then ask: why is the term

```

exp : Expr
exp = (nilE [nat 0] = nat 1) !E (nat 0 + E nat 1)

```

welltyped? The answer given by $WtExpr$ is

```

wt-exp : WtExpr exp TOption
wt-exp = lift-wt-array (ok-lookup wta wt+)
  where
    wta : WtExpr (nilE [nat 0] = nat 1) TArray
    wta = lift-wt-array
      (ok-ins (lift-wt-array ok-nil)
        (lift-wt-nat 1) (lift-wt-nat 0))
    wt+ : WtExpr (nat 0 + E nat 1) TNat
    wt+ = lift-wt-sum (ok-sum
      (lift-wt-nat 0) (lift-wt-nat 1))

```

The $lift-wt-*$ functions play the same role in $WtExpr$ as inn does in $Expr$; however rather than using the generalized approach of a series of disjoint sums we bundle the tag and recursion into a single constructor for each language component. Evaluation displays a similar symmetry

```

eval-expr : (nilE [nat 0] = nat 1) !E (nat 0 + E nat 1)
  → (nilE [nat 0] = nat 1) !E nat 1
eval-expr = step [] (stepi (step+ stepv))

```

What does the proof that $(nilE [nat 0] = nat 1) !E nat 1$ is welltyped look like? We can compute it by invoking

```

preservation eval-expr wt-exp

```

which evaluates to

```

lift-wt-array
(ok-lookup
  (lift-wt-array
    (ok-ins (lift-wt-array ok-nil) (lift-wt-nat 1) (lift-wt-nat 0)))
  (lift-wt-nat 1))

```

7. Related Work

Independent and concurrently with our work, Delaware, et al. [7] developed a solution to modular meta-theory in Coq. Both their approach and ours relies on the principle of representing data types as functors; however they have chosen to express inductive types using Church encodings and recursive evaluation using Mendler algebras, which requires some extra sophistication. Here we express types as data members of the family of polynomial functors and apply recursive evaluation directly. Their approach presented is further along and has shown the important level of robustness required by most languages while there are more unanswered questions regarding the method presented here.

8. Conclusion and Future Work

We should ask if we have accomplished the goal that we set out with. The language $Expr$ was given componentwise and the boiler-plate necessary to wrap each welltyping and step relation is minimal. The proof of type preservation was almost immediate, requiring only an invocation of previously defined proofs for each component. Moreover there is no copy and paste necessary and the repetitive components should be automatically producible given a sophisticated macro system where terms can be inspected by name—set equality is non-deterministic—rather than value.

Using Agda as a proof language, although convenient, leaves the question of consistency open. We regard this as a minor problem and hope that our implementation would port to Coq. A more pertinent problem is the definition of *preservation* for $Expr$ —Agda is unable to prove termination and we plan to address this soon.

The language presented is quite simple, unable to express even Euclid's algorithm, and the method of polynomial functor's used to express $Expr$ precludes the possibility of first class function types which are critical for functional programming. Various solutions to this problem have been proposed [5] and the area of recursion schemes is rich [6]. A real world language calls for much heavier sophistication, but the ideas presented here are new and their reach is open to question and requires further exploration.

References

- [1] Aydemir, et al. *Mechanized Metatheory for the Masses: The PoplMark Challenge*. In International Conference on Theorem Proving in Higher Order Logics (TPHOLs), August 2005.
- [2] Grant Malcom. *Algebraic data types and program transformations*. Ph.D. thesis, Department of Computing Science, Groningen University, 1990.
- [3] Luc Duponcheel. *Using catamorphisms, subtypes and monad transformers for writing modular functional interpreters*. Utrecht University, 1995.
- [4] Ulf Norell. *Dependently Typed Programming in Agda*. In Proceedings of the 4th international workshop on Types in language design and implementation (TLDI '09). ACM, New York, NY, USA, 1-2.
- [5] Erik Meijer and Graham Hutton. *Bananas in Space: Extending Fold and Unfold to Exponential Types*. In Proceedings of the seventh international conference on Functional programming languages and computer architecture (FPCA '95). ACM, New York, NY, USA, 324-333.
- [6] Tarmo Uustalu, et al. *Recursion Schemes from Comonads*. *Nordic J. of Computing* 8, 3 (September 2001), 366-390.
- [7] Benjamin Delaware, Bruno C. d. S. Oliveira, Tom Schrijvers. *Meta-Theory à la Carte*, unpublished manuscript, July, 2012.